Complexity of Data Flow Analysis for Non-Separable Frameworks

Bageshri (Sathe) Karkare*
Dept. of Computer Science & Engg.,
Indian Institute of Technology, Bombay
Mumbai, India
bageshri@cse.iitb.ac.in

Uday Khedker
Dept. of Computer Science & Engg.,
Indian Institute of Technology, Bombay
Mumbai, India
uday@cse.iitb.ac.in

Abstract— *The complexity of round robin iterative data flow analysis has been traditionally defined as $1+d$ where d is the depth of a control flow graph. However, this bound is restricted to bit vector frameworks, which by definition, are separable. For non-separable frameworks, the complexity of analysis is influenced by the interdependences of program entities, hence the bound of $1+d$ is not applicable. This motivates the need for capturing the interdependences of entities to define a general complexity measure.*

We propose Degree of dependence δ *which quantifies the effect of non-separability on the complexity of analysis for a particular problem instance. We define the complexity bound of $1+\delta+d$ which explains the complexity of round robin analysis of general non-separable data flow problems. Like d, δ is a theoretical concept useful for understanding the complexity rather than estimating it. In bit vector frameworks the bound $1+\delta+d$ reduces to $1+d$ due to $\delta=0$. Apart from being general, our bound is also precise, as corroborated by empirical results.*

Index Terms— Data flow analysis, Complexity, Constant propagation

I. Introduction

For a given instance of a data flow framework, the complexity of performing data flow analysis depends on a combination of the properties of program structure (in particular, loops and their complex arrangements), separability and boundedness of flow functions, height of the lattice, and the order of traversal over the control flow graph. Among the general complexity measures for data flow analysis ([1], [3], [7], [12], [13], [17], [19], [22]) the number of iterations required for convergence of round robin iterative method has been explored in details. The traditional complexity bound of round robin data flow analysis for bit vector frameworks has been defined as $1+d$ iterations ([1], [7], [12]), where d is the depth of the control flow graph which is defined as the maximum number of back edges in any acyclic path[1].

Depth d is not independent of the problem instance since it captures the structure of control flow. However, it does not depend on the program statements. Since bit-vector problems are separable 2-bounded problems, program statements do not play a significant role in increasing the complexity. Hence their complexity can be reasonably explained using only the depth d. In non-separable frameworks like constant propagation ([1], [7], [18]), faint variables analysis ([8], [15]), type inferencing of flow sensitive types [11], and points-to analysis ([4], [10]) the complexity of analysis is influenced by the interdependences among program entities, which in turn are heavily influenced by the placement of program statements, apart from depth. Complexity of analyzing such problems has been observed to be proportional to number of program entities, but no strict bound has been reported.

We define an *Entity Dependence Graph* to capture the non-separable information flows in the given program instance. We propose δ, the *degree of dependence* as a measure of effect of non-separability on complexity. We use δ to define a complexity bound of $1+\delta+d$ which is the first ever realistic explanation of the complexity of data flow analysis for non-separable frameworks. Note that like d, δ is a purely theoretical concept which is used for explanation rather than for estimation of complexity. Our complexity bound is uniformly applicable to a large class of data flow frameworks. In particular, the separability of bit vector frameworks can be viewed as a special case of non-separability, due to which $\delta=0$ and the bound $1+\delta+d$ reduces to $1+d$. Apart from being general, our measure is precise. Our empirical measurements for constant propagation and faint variables analysis corroborate this.

The rest of the paper is organized as follows: Section II reviews the theory of data flow analysis. Section III introduces the entity dependence graph. Sec-

*Supported by Infosys Technologies Limited, Bangalore, under Infosys Fellowship Award.

[1] For bidirectional frameworks, this has been generalized to $1+w$ ([3], [12], [13]) where w is the maximum width of any information flow path.

tion IV defines the degree of dependence and derives the bound of $1+\delta+d$. Section V provides empirical measurements.

II. Data Flow Frameworks

A data flow framework D is a tuple $\langle L, \sqcap, F\rangle$ ([1], [6], [7], [9], [12], [14], [18]), where L is a lattice with meet $\sqcap$ and F is the set of flow functions $L \mapsto L$. L is the set of data flow values with the partial order $\sqsubseteq$ induced by $\sqcap$ and has a top element $\top$ and a bottom element $\bot$. An instance I_D, of a data flow framework D, is a pair $\langle G, M\rangle$ where G is a control flow graph (CFG) and M is a mapping from the nodes/edges of the CFG to the functions in F. Note that D defines only the structure of L, it is actually instantiated by I_D depending upon the actual number of entities (eg. variables or expressions etc.).

A. Properties of Lattices

For a lattice L, a *strictly descending chain* is a sequence $v_0 \sqsupset v_1 \sqsupset \ldots \sqsupset v_n$ such that $v_i \in L$. Analogously, a *strictly ascending chain* is $v_n \sqsubset v_{n-1} \sqsubset \ldots \sqsubset v_0$. The lattices in which all strictly descending chains are finite have been called *bounded* in ([9], [14]) and *finite* in [7]. The lattices in which both strictly descending as well as strictly ascending chains are finite have been called *finite* in ([12], [13]) and *complete* in ([21]). In the rest of this paper, by bounded lattices we mean complete lattices.

The overall lattice L is a product of component lattices $\widehat{L}_1 \times \widehat{L}_2 \times \cdots \times \widehat{L}_\xi$, where $\widehat{L}_i$ is a lattice of the values of data flow properties of an individual entity α_i, and ξ is the number of entities. The $\top, \bot \in L$ are tuples $\langle \widehat{\top}_1, \widehat{\top}_2, \ldots, \widehat{\top}_\xi\rangle$ and $\langle \widehat{\bot}_1, \widehat{\bot}_2, \ldots, \widehat{\bot}_\xi\rangle$. In the case of available expressions analysis, for a given expression e, $\widehat{\top}$ is $\{e\}$ and $\widehat{\bot}$ is $\emptyset$. $\sqcap$ and $\sqsubseteq$ are $\cap$ and $\subseteq$ respectively. In the case of reaching definitions analysis, for a given definition $di : x = e$, $\widehat{\top}$ is $\emptyset$ and $\widehat{\bot}$ is $\{di\}$. Further, $\sqcap$ and $\sqsubseteq$ are $\cup$ and $\supseteq$ respectively.

The height of a lattice is the maximum number of $\sqsubset$ or $\sqsupset$ in any strict chain. Let $\widehat{H}_i$ denote height of $\widehat{L}_i$. Often all $\widehat{L}_i$ are same and the height H of the overall lattice L is:

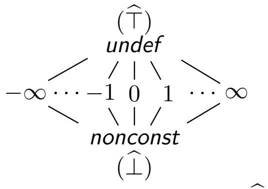
$$H = \widehat{H}_i \times \xi \tag{1}$$

B. Properties of Flow Functions

Flow functions are *monotonic* ([1], [6], [7], [9], [12], [14], [18]) if:

$$\forall f \in F, \forall x, y \in L \; : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \tag{2}$$

If the flow functions $f : L \mapsto L$ in F are tuples of functions $f = \langle \widehat{h}_1, \widehat{h}_2, \cdots, \widehat{h}_k\rangle$ such that $\widehat{h}_i : \widehat{L}_i \mapsto \widehat{L}_i$, then the framework is *separable* ([12], [13]) in that the functions on one component lattice operates independently of others.



(a) Component Lattice $\widehat{L}$

f^+	$\widehat{\top}$	c_2	$\widehat{\bot}$
$\widehat{\top}$	$\widehat{\top}$	$\widehat{\top}$	$\widehat{\bot}$
c_1	$\widehat{\top}$	$c_1 + c_2$	$\widehat{\bot}$
$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$

(b) Flow function for +

$\sqcap$	$\widehat{\top}$	c_2	$\widehat{\bot}$
$\widehat{\top}$	$\widehat{\top}$	c_2	$\widehat{\bot}$
c_1	c_1	$c_1 = c_2 \Rightarrow c_1$ $c_1 \neq c_2 \Rightarrow \widehat{\bot}$	$\widehat{\bot}$
$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$

(c) Confluence operation

Fig. 1. Constant propagation framework.

The flow functions are *k-bounded* if their *loop closures* ([12], [13], [17]) (also called *fastness closures* in [6], [7], [21]) are bounded by constant k. Let $f^{j+1} = f \circ f^j$ and f^0 be the identity function. f^j represents the flow function corresponding to the path containing j traversals over a loop. k-boundedness requires that

$$\exists\, k \geq 1 \; s.t. \; \forall\, f \in F : \; f^0 \sqcap f^1 \sqcap f^2 \sqcap \cdots = \prod_{i=0}^{k-1} f^i \tag{3}$$

k-boundedness implies that though a program contains infinite paths in the presence of loops, a finite number of paths containing upto $k-1$ unfoldings of loops are sufficient for convergence of data flow analysis.

The computation of $f^0 \sqcap f^1 \sqcap f^2 \sqcap \cdots \sqcap f^i$ represents the *greatest lower bound* (*glb*) of the results of first i applications of f. Thus it follows a descending chain in L. Hence bounded lattices imply bounded loop closures.

For separable frameworks, values in $\widehat{L}$ change simultaneously due to independence of the component lattices. It is easy to verify that for a k-bounded framework,

$$k \leq \begin{cases} \widehat{H}+1 & \text{if the framework is separable} \\ H+1 & \text{if the framework is non-separable} \end{cases} \tag{4}$$

Bit vector frameworks have separable flow functions which operate on $\widehat{L}$ with $\widehat{H} = 1$. Further, since all flow functions in bit vector frameworks are either constant functions or identity, $f^2(x) = f(x)$ implying that k is 2 for bit vector frameworks.

C. Examples of Non-Separable Frameworks

In this section, we briefly introduce constant propagation and faint variables analysis which are used as running examples in the paper.

Constant propagation [1], [7], [18] identifies variables which hold a fixed constant value and replaces them by this value. Lattice $\widehat{L}$ of data flow values

- Lhs_n : Singleton set containing LHS variable of statement n.
- Rhs_n : Set of variables used in RHS expression of statement n which is an assignment statement. For non-assignment statements like *print* and for function calls, this set is empty.
- Use_n : Set of variables used in statement n, where n is either a non-assignment statement or a function call.
- $\mathit{FaintGen}_n/\mathit{FaintKill}_N$: Set of variables whose faintness is generated/killed by statement n.
- $\mathit{FaintIn}_n/\mathit{FaintOut}_n$: Faintness information at entry/exit of node n.

$$
\begin{aligned}
\mathit{FaintGen}_n &= \{x \mid x \in \mathit{Lhs}_n, x \notin \mathit{Rhs}_n\}\\
\mathit{FaintKill}_n &= \{x \mid x \in \mathit{Rhs}_n, y \in \mathit{Lhs}_n, y \notin \mathit{FaintOut}_n\}\\
&\quad \cup \{x \mid x \in \mathit{Use}_n\}\\
\mathit{FaintIn}_n &= (\mathit{FaintOut}_n - \mathit{FaintKill}_n) \cup \mathit{FaintGen}_n\\
\mathit{FaintOut}_n &= \begin{cases} \top & n \text{ is exit node}\\ \displaystyle\bigcap_{s \in succ(n)} \mathit{FaintIn}_s & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 2. Data flow equations to determine faint variables

Constant Propagation

- w is 2 at the exit of 4. No variable is constant at any other point.
- $\widehat{H} = 2$, $H = 8$, $w = 3$
- Predicted bound : 25 iterations
- Actual iterations : 9
- If the assignments in nodes 5 and 7 are exchanged, $\widehat{H}$, H, and d (hence predicted bound) remain same. Actual number of iterations reduces to 5.

Faint Variables Analysis

- Due to the *print* statement, no variable is faint at any point.
- $\widehat{H} = 1$, $H = 4$, $w = 3$
- Predicted bound : 13 iterations
- Actual iterations : 7
- If the assignments in nodes 5 and 7 are exchanged, $\widehat{H}$, H, and d (hence predicted bound) remain same. Actual number of iterations reduces to 5.

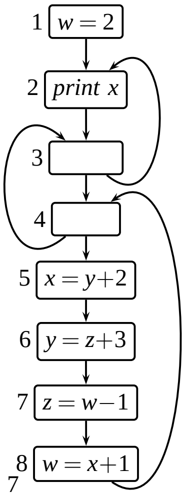


Fig. 3. Complexity bounds for non-separable analyses are very loose.

for a variable is shown in Figure 1(a). *undef* denotes an undefined value and forms the $\widehat{\top}$ whereas *nonconst* denotes a non-constant value and forms the $\widehat{\bot}$. All constants c_i, $-\infty \le c_i \le \infty$ are ordered such that $c_i \sqsubset \widehat{\top}$ and $\widehat{\bot} \sqsubset c_i$. Observe that this lattice has an infinite number of elements but a finite height: $\widehat{H} = 2$ and $H = 2 \times \xi$. Confluence operation for constant propagation is shown in Figure 1(c). Flow functions which influence the data flow properties of variable correspond to statements such as $x = y$, $x = y + z$, $x = 3$ or $x = \mathit{read}()$. For the first two statements, the flow functions compute the data flow value of x from the data flow values of the variables appearing in the right hand side. For the last two, the data flow values of x are c_3 and *nonconst* respectively. Figure 1(b) shows the flow function corresponding to a statement $x = y + z$. Observe that it is of the form $(\widehat{L}_x \times \widehat{L}_y) \mapsto \widehat{L}_z$ rather than $\widehat{L} \mapsto \widehat{L}$. Due to non-separability, its loop closure bound k is $H + 1$ and is not constant.

Faint variables analysis [8], [15] is a more general variant of live variables analysis in that it computes the transitive closure of dead variables. A variable is faint if it is dead or if it is only used to compute new values of faint variables. Variables which are not faint are *strongly live* [20].

Data flow equations to identify faint variables are given in Fig. 2. For a variable v, $\widehat{\top}$ is $\{v\}$, $\widehat{\bot}$ is $\emptyset$ and $\widehat{H} = 1$. Though these data flow properties can be represented using bit vectors, the flow functions are non-separable. Hence this is not a bit vector framework.

D. Simplistic Generalization of Complexity Bound

Using only the properties of framework, a simplistic generalization of the complexity bound can be formulated as follows: In round robin method, in the worst case, each iteration may change exactly one data flow value. For general frameworks, at most H new values may be generated at any program point. Since computation of each new value may require at most d additional iterations, the total number of iterations is $1 + (d \times H)$. In a k-bounded framework, at most $(k-1)$ new values can be generated at a program point. Hence $(k-1)$ can replace H giving a possibly stricter bound of $1 + d \times (k-1)$ (see Equation 4). This has already been conjectured in [12]. However, since k may not be known, we continue to use $1 + (d \times H)$. For separable frameworks, it is $1 + (d \times \widehat{H})$.

Example 1: Consider constant propagation and faint variables analysis for the CFG in Figure 3. Observe that the actual number of iterations is much smaller than the predicted bound. Besides, a rearrangement of statements causes a significant variation in the number of iterations and the predicted bound is insensitive to this change.

Separability guarantees that the data flow properties of all entities are independent of each other whereas handling non-separability requires the assumption that in the worst case, the data flow property of every entity depends on the data flow properties of all entities. We view these two extreme cases of dependence as two extremes of the same continuous spectrum by modeling the exact cause of non-separability explicitly and by defining the degree of dependence as a measure of non-separability. The degree of dependence is defined for a particular problem instance using the *Entity Dependence Graph* which we introduce in the next section.

III. Entity Dependence Graph

Let the flow function corresponding to a statement s be f. Let dfpmod_f denote the set of entities which occur in s and whose data flow properties are computed by f. Note that for the flow of information, this computation should potentially compute a non-$\widehat{\top}$ value because information flow consists of propagating non-$\widehat{\top}$ value. Let dfpuse_f denote the set of entities which occur in s and whose data flow properties are used by f.

Formally, an *entity dependence graph* (*EDG*) is a directed graph $G_{edg} = \langle N_{edg}, E_{edg} \rangle$. N_{edg} is the set of entities defined as follows:

$N_{edg} = \{\alpha_s \mid \alpha \in \mathit{dfpmod}_f,\ f$ is a flow function for statement $s\}$

α is the name of an entity whereas α_s represents its instantiation for statement s. α_s^v denotes entity α_s with value v. Different instances of the same entity may have to be created in the following situations:

1) When a data flow problem requires such a distinction. For example, reaching definitions analysis requires enumeration of all definitions of each variable reaching a program point.
2) When different instances of the same entity reaching a program point have different dependence relations and hence different influences on the complexity.

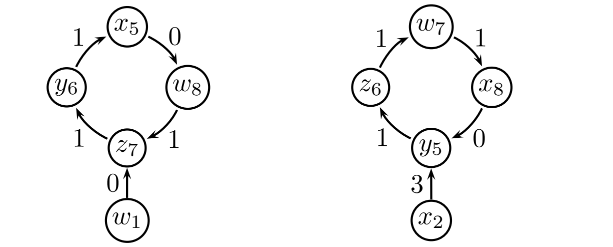
E_{edg} is the set of edges between entities. An edge $\alpha_i \rightarrow \beta_j$ indicates that the data flow property of entity β_j directly depends on the data flow property of α_i, where i and j may not be adjacent in CFG. $\alpha_i^v \rightarrow \beta_j^u$ indicates that value u of β_j is due to the direct influence of value v of α_i. Constructing a dependence $\alpha_i \rightarrow \beta_j$ requires:

- identifying the presence of a flow function f at a program point p associated with statement j such that $\beta_j \in \mathit{dfpmod}_f$ and $\alpha \in \mathit{dfpuse}_f$, and
- discovering the instance α_i reaching s for $\alpha \in \mathit{dfpuse}_f$.

For information propagation, each *EDG* must contain some entry node i.e. a node with no predecessors: Some entity must change from $\widehat{\top}$ to non-$\widehat{\top}$ independently, otherwise all entities would remain $\widehat{\top}$. In the latter situation, data flow analysis need not be performed. For separable frameworks EDG-edges can exist only between instances of the same entity because data flow properties of an entity change independently of other entities.

Example 2: Some *EDG*s for the CFG in Figure 3 are as described below:

Available Expressions Analysis: In a bit vector framework, flow functions are either constant functions or identity and all entities are independent of others. Hence *EDG* has no edges. For available expressions analysis, an entity α_s represents the fact that expression α is killed in statement s. For the instance in Figure 3, the set of *EDG*-nodes is $\{(y+2)_6, (z+3)_7, (w-1)_1, (w-1)_8, (x+1)_5\}$ and the set of edges is $\emptyset$.



(a) *EDG* for Constant propagation problem

(b) *EDG* for Faint variables problem

Fig. 4. Entity Dependence Graphs for the CFG in Figure 3. Edges are labeled with edge-weights.

Constant Propagation: For constant propagation, an entity is identified by a variable name and statement number associated with a definition of that variable. The resulting *EDG* is shown in Fig. 4(a). Nodes w_8 and w_1 correspond to definitions of w in statements 8 and 1 respectively. A reaching definitions analysis using the renamed definitions can identify the *EDG*-edges. An edge label represents weight of an edge (defined in Section IV).

Faint Variables Analysis: Figure 4(b) shows the *EDG*. The uses of x in statements 2 and 8 are treated as different entities denoted by x_2 and x_8 respectively. An assignment $s : a = b$ adds dependence edges $a_i \rightarrow b_s$ such that a_i is live at exit of s. Identifying dependence edges requires live variables analysis with renamed uses.

An *EDG*-path $\alpha_i \dashrightarrow \beta_j$ represents a transitive influence of α_i on β_j. Cycles in *EDG* represent self-dependences. Interestingly, a cyclic *EDG*-path may differ from a cyclic CFG-path. For example, in the *EDG* for constant propagation in Figure 4(a), path $(z_7, y_6, x_5, w_8, w_7)$ captures the cyclic dependence among entities z_7, y_6, x_5, w_8, and z_7. The CFG in Figure 3 contains many cycles not necessarily coinciding with this *EDG*-cycle. Further, each edge in the *EDG*-cycle corresponds to a cyclic path involving the back edge 8 → 5 in the CFG.

Observe that the dependence captured by *EDG* is different from the dependences explored in past. Control dependence[5] captures the dependence of execution of a statement on other statements. Data dependences[16], Def-Use and Use-Def chains[1], SSA edges[2] etc. capture the dependence of a statement on the data computed in other statements. These traditional dependences do not capture the dependence of data flow properties and hence cannot be used for complexity analysis.

IV. Defining Complexity Using The Degree of Dependence

For an *EDG* edge $\alpha_i \rightarrow \beta_j$, edge-weight denoted as $Wt(\alpha_i \rightarrow \beta_j)$ is defined as the maximum number of back edges in any acyclic path from stmt i to stmt j for forward problems, and from stmt j to stmt i for backward problems[2].

Example 3: All edges in the *EDG* shown in Figure 4 are labeled with their weights. In the *EDG* for faint variables analysis, edge $x_2 \rightarrow y_5$ has weight 3 since the path from stmt 5 to stmt 2 contains three back-edges. Edge $x_8 \rightarrow y_5$ has depth 0 because of the back-edge-free path from stmt 5 to stmt 8.

The *weight* of *EDG*-path $\alpha_i \rightarrowtriangle \beta_j$ denoted $Wt(\alpha_i \rightarrowtriangle \beta_j)$, is defined as the sum of weights of the edges along the path. It is defined only for an acyclic path with the relaxation that α_i and β_j may be same. The *degree of dependence* of an *EDG*-path $\alpha_i \rightarrowtriangle \beta_j$, denoted $\Delta(\alpha_i \rightarrowtriangle \beta_j)$ is defined according to the structure of the path as follows:

- If the path $\alpha_i \rightarrowtriangle \beta_j$ is acyclic, $\Delta(\alpha_i \rightarrowtriangle \beta_j) = Wt(\alpha_i \rightarrowtriangle \beta_j)$.
- If the path $\alpha_i \rightarrowtriangle \beta_j$ is acyclic except that β_j is same as α_i, then a change in α_i has to be propagated to all entities along the path, including α_i. Doing so once may require $Wt(\alpha_i \rightarrowtriangle \beta_j)$ additional iterations. This may change α_i further. The maximum number of changes is $\widehat{H}$, and

  $$\Delta(\alpha_i \rightarrowtriangle \beta_j) \quad = \quad \widehat{H} \times Wt(\alpha_i \rightarrowtriangle \beta_j) \qquad (5)$$

  In case of overlapping cycles, the maximum value of $\Delta(\alpha_i \rightarrowtriangle \alpha_i)$ is considered.
- If the path $\alpha_i \rightarrowtriangle \beta_j$contains m non-overlapping cycles connected by $m-1$ acyclic segments, it has the following structure:

  $$\alpha_i \rightarrowtriangle \gamma_1 \rightarrowtriangle \gamma_1 \rightarrowtriangle \gamma_2 \rightarrowtriangle \gamma_2 \rightarrowtriangle \cdots \gamma_m \rightarrowtriangle \gamma_m \rightarrowtriangle \beta_j \quad (6)$$

  Then $\Delta(\alpha_i \rightarrowtriangle \beta_j)$ is defined as:

$$\begin{aligned}\Delta(\alpha_i \rightarrowtriangle \beta_j) \quad = \quad & \sum_{n=1}^{m} \Delta(\gamma_n \rightarrowtriangle \gamma_n) + \sum_{n=1}^{m-1} \Delta(\gamma_n \rightarrowtriangle \gamma_{n+1}) \\ & + \Delta(\alpha_i \rightarrowtriangle \gamma_1) + \Delta(\gamma_m \rightarrowtriangle \beta_j) \qquad (7)\end{aligned}$$

$\Delta(\alpha_i \rightarrowtriangle \beta_j)$ computes the maximum number of iterations required to propagate the influence of a change in the value of the data flow property of α_i on the value of the data flow property of β_j along the path. Let α_0 denote an entry node of the *EDG* for an instance I_D for a given D. The degree of dependence of I_D is denoted by δ and is defined as follows:

$$\delta = \max(\Delta(\alpha_0 \rightarrowtriangle \beta_j)), \text{ for any } \alpha_0 \text{ and } \beta_j \qquad (8)$$

In the case of practical non-separable problems like constant propagation and faint variables analysis, δ can be made more precise. We define the entity dependence in a non-separable framework to be monotonic if

$$\forall \alpha_i^v \rightarrow \beta_j^u,\ ht(u) \geq ht(v) \qquad (9)$$

$ht(v)$ denotes the height of v in the component lattice and is defined as the length of a longest descending chain from $\widehat{\top}$ to v. For $Y = f(X)$, let $X, Y \in L$ be $\langle \widehat{X}_1, \widehat{X}_2, \ldots, \widehat{X}_\xi \rangle$ and $\langle \widehat{Y}_1, \widehat{Y}_2, \ldots, \widehat{Y}_\xi \rangle$. Let $\widehat{X}_i$ determine the value of $\widehat{Y}_j$. Then the possible values of $\widehat{Y}_j$ are limited to those at same height as $\widehat{X}_i$ or with higher height (effectively lower in lattice).

Example 4: For constant propagation, let c denote any constant. Then the longest strictly descending chain in $\widehat{L}$ is $\widehat{\top} \sqsupset c \sqsupset \widehat{\bot}$ with heights of elements $(0,1,2)$ respectively. For some $\alpha_i^v \rightarrow \beta_j^u$, condition (9) can only be violated if $\langle v, u \rangle$ are $\langle c, \widehat{\top} \rangle$, $\langle \widehat{\bot}, c \rangle$ or $\langle \widehat{\bot}, \widehat{\top} \rangle$. However this is not possible implying that constant propagation has monotonic entity dependence. For faint variables analysis, the corresponding chain is $\widehat{\top} \sqsupset \widehat{\bot}$. It can be easily verified that the entity dependence is monotonic.

Theorem 1:. For a data flow framework with monotonic entity dependence, the term $\displaystyle\sum_{n=1}^{m} \Delta(\gamma_n \rightarrowtriangle \gamma_n)$ in Equation (7) reduces to $\widehat{H} \times \max(Wt(\gamma_n \rightarrowtriangle \gamma_n))$.

Proof : In path structure (6), the multiplication factor $\widehat{H}$ in $\Delta(\gamma_n \rightarrowtriangle \gamma_n)$ (Equation 5) is required only if the flow is $\gamma_n^v \rightarrowtriangle \gamma_n^u$ such that $ht(u) > ht(v)$. Due to monotonic entity dependence, the values computed in each cycle will get progressively limited and instead of $\widehat{H}$ changes in each cycle, the changes will get distributed over m cycles. Let the number of changes in cycle $\gamma_n \rightarrowtriangle \gamma_n$ be c_n. Then we need to maximize the term

$$\sum_{n=1}^{m} c_n \times Wt(\gamma_n \rightarrowtriangle \gamma_n) \text{ such that } \sum_{n=1}^{m} c_n = \widehat{H}$$

The constraint on c_n defines a hyperplane and a maximum occurs at some extreme point of the plane. Hence the maximum value of the summation is $\widehat{H} \times \max(Wt(\gamma_n \rightarrowtriangle \gamma_n))$. □

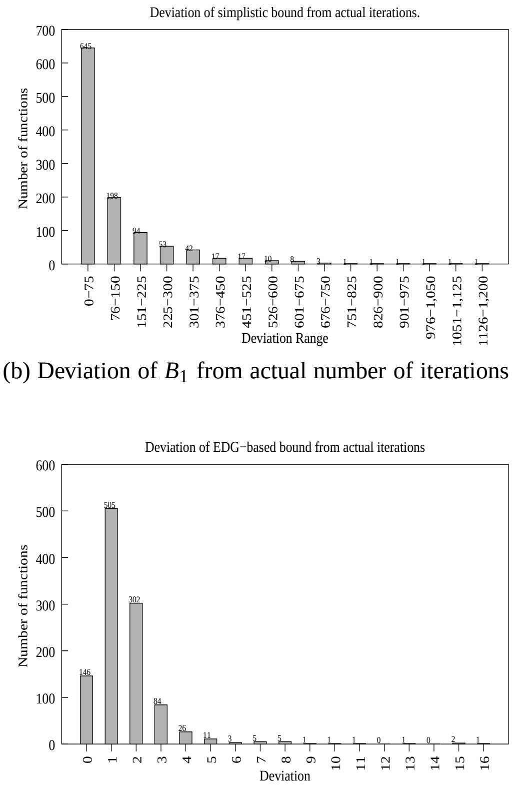
Theorem 2:. A round robin data flow analysis of an instance I_D of a data flow framework D would converge in at most $1 + \delta + d$ iterations.

Proof : The first change in every α_0 must happen in the very first iteration because it does not depend on any other entity. From Equation (8), the maximum number of iterations required for computing the final value of the data flow property of any entity α_i is $1 + \delta$. However, it is computed at statement i and also needs to be propagated to those statements in the CFG for which this entity may not occur in $dfpuse_f$. This may requires at most d additional iterations taking the bound to $1 + \delta + d$. □

[2] for bidirectional problems, the notion of *width* [13] must be used.

Benchmark	$\#F$	$\#V$	d	δ	B_1	B_2	I
arith_coder	29	40.90	1.31	1.83	117.76	4.14	2.90
186.crafty	38	56.39	1.39	2.08	177.84	4.47	2.87
whetstone	2	9.00	1.00	1.50	19.00	3.50	2.50
kexis	11	33.27	1.09	1.55	76.45	3.64	2.55
175.vpr	157	37.81	1.53	2.09	137.83	4.62	3.19
181.mcf	20	25.65	1.35	1.50	69.30	3.85	2.30
197.parser	213	11.78	1.37	1.27	39.97	3.64	2.53
164.gzip	56	36.54	1.36	2.14	111.25	4.50	2.64
gsm	32	29.97	1.16	1.59	73.25	3.75	2.59
300.twolf	110	46.26	1.67	1.83	160.58	4.50	2.94
256.bzip2	25	36.00	1.52	2.16	135.96	4.68	2.92
254.gap	401	43.10	1.27	2.20	124.63	4.48	2.63

(a) Summary of complexity computation for various benchmarks: $\#F$ is the number of functions, $\#V$ is the average number of variables. All other values are average values: B_1 is the predicted number of iterations using $1+d\times H$, B_2 is the predicted number of iterations using $1+\delta+d$, I is the actual number of iterations.



(b) Deviation of B_1 from actual number of iterations

(c) Deviation of B_2 from actual number of iterations

Fig. 5. Empirical measurements

Corollary 1:. For bit vector frameworks, the bound in Theorem (2) reduces to $1+d$.

Proof : For bit vector frameworks, *EDG* does not contain any edges. Hence $\delta = 0$. □

Example 5: In the *EDG* for constant propagation in Figure 4(a), $\alpha_0 = d_1$. Since $\widehat{H} = 2$, $\Delta(\alpha_0 \twoheadrightarrow \beta_j)$ for each β_j in $\{w_1, z_7, y_6, x_5, w_8\}$ is $\{0,6,6,6,6\}$ respectively. Hence $\delta = 6$. Depth of the CFG is $d = 3$. Thus complexity bound $(1+\delta+d)$ is 10. This compares well with the actual number of iterations which is 9.

In the *EDG* for faint variables analysis in Figure 4(b), $\alpha_0 = a_2$. Since $\widehat{H} = 1$, $\Delta(\alpha_0 \twoheadrightarrow \beta_j)$ for each β_j in $\{x_2, y_5, z_6, w_7, x_8\}$ is $\{0,6,6,6,6\}$ respectively. Hence $\delta = 6$. Using $d = 3$, we get the overall complexity bound 10. The actual number of iterations is 7.

V. Empirical Measurements

We have implemented the complexity computation and round robin data flow analysis for constant propagation as well as faint variables analysis to measure the precision of the bound $1+\delta+d$ for practical programs. Our prototype implementation uses XSB Prolog[3]. The input is obtained by translating gimple IR produced by gcc version 4.0.0.

We have tested our implementation on SPEC-2000 C benchmark programs, whetstone and digital signal processing benchmarks such as gsm. Since the implementation is restricted to intraprocedural level, we make conservative assumptions for values of global variables used in expressions. Since our goal is not to use the information for transformation but to measure the precision of our bound, aliases are ignored. We have also computed the simplistic bound (Section 5) $1+d\times H$ where $H = 2\times|V|$ for constant propagation.

Figure 5(a) summarizes the results for constant propagation presenting averages of various numbers. The table excludes the information of acyclic programs since for acyclic programs, the bounds are trivially 1. Clearly, our bound is very close to the actual number of iterations. Figures 5(b) and (c) plot the number of functions against the deviations of the bounds from the actual number of iterations. Clearly, the bound $1+\delta+d$ has very small deviation (0 to 2) for a large number (87.11%) of programs. The bound $1+d\times H$ has large deviations for most of the programs. In particular, small deviation (0 to 2) is found in only 1.28% cases. The deviations of both

[3] Available from `http://xsb.sourceforge.net`

bounds is 0 for acyclic program and we have excluded them.

VI. Conclusions and Future Work

Non-separability of data flow framework is a dominant factor influencing the complexity of round robin data flow analysis. The existing theory accounts for lattice height and loop closure bounds in determining complexity bounds. However, it fails to capture the exact role played by non-separability. This paper proposes the concept of degree of dependence which is a more precise measure since (a) It uses the height of the component lattice instead of overall lattice while considering the cumulative effect only for interdependent entities, and (b) It distinguishes the cyclic CFG paths from the cyclic dependences. Apart from precision, these distinctions facilitate generality by placing separable and non-separable frameworks on the same continuous spectrum. This provides a uniform explanation of the phenomena observed in a large class of practical instances of a variety of data flow frameworks.

We would like to extend this work to point-to analysis [4], [10] of C and C++ where new entity dependences are discovered during analysis due to pointer indirections. Yet another interesting direction of future work is to explore the use of entity dependence graph for performing data flow analysis.

Acknowledgments

We wish to thank Amitabha Sanyal, Supratim Biswas and Amey Karkare for their useful comments. Garima Lahoti helped in the prototype implementation.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[3] Dhananjay M. Dhamdhere and Uday P. Khedker. Complexity of bi-directional data flow analysis. In *POPL '93*, pages 397–408, New York, NY, USA.

[4] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94*, pages 242–256, New York, NY, USA.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[6] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. In *POPL '75*, pages 22–34, New York, NY, USA, 1975.

[7] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., NY, USA, 1977.

[8] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Inf.*, 24(6):679–694, 1987.

[9] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–318, 1977.

[10] Aditya Kanade, Uday P. Khedker, and Amitabha Sanyal. Heterogeneous fixed points with application to points-to analysis. In *APLAS'05*, pages 298–314.

[11] U. P. Khedker, D. M. Dhamdhere, and A. Mycroft. Bidirectional data flow analysis for type inferencing. *Computer Languages, Systems and Structures*, 29(1-2):15–44, 2003.

[12] Uday P. Khedker. Data flow analysis. In *The Compiler Design Handbook*, pages 1–59. CRC Press, 2002.

[13] Uday P. Khedker and Dhananjay M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM TOPLAS*, 16(5):1472–1511, 1994.

[14] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73*, pages 194–206, New York, NY, USA, 1973.

[15] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *PLDI '94*, pages 147–158, New York, NY, USA, 1994.

[16] David J. Kuck, R. H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *POPL'81*, pages 207–218.

[17] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Inf.*, 28:121–163, 1990.

[18] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[19] Robert Muth and Saumya Debray. On the complexity of flow-sensitive dataflow analyses. In *POPL '00*, pages 67–80, New York, NY, USA, 2000.

[20] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[21] H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In *CC'98*, pages 109–127, 1997. LNCS 1383, Springer-Verlag.

[22] Hanne Riis Nielson and Flemming Nielson. Bounded fixed point iteration. In *POPL '92*, pages 71–82, New York, NY, USA, 1992.